Exam in Operating Systems (EDAF35)
2018-05-29, 14:00–19:00

Inga hjälpmedel! No external resources allowed!

Examiner: Flavius Gruian, tel 046 2224518

25 out of 50p are needed to pass the exam.
You may answer in English/på svenska.

1. (6p) Define the following terms (1–2 sentences each):
   (a) (1p) microkernel ***Answer*** *See lecture slides for Chapter 2.*
   (b) (1p) round robin scheduling ***Answer*** *See lecture slides for Chapter 6.*
   (c) (1p) thrashing ***Answer*** *See lecture slides for Chapter 9.*
   (d) (1p) double buffering ***Answer*** *See lecture slides for Chapter 13.*
   (e) (1p) memory-mapped I/O ***Answer*** *See lecture slides for Chapter 13.*
   (f) (1p) hypervisor ***Answer*** *ee lecture slides for Chapter 16.*

2. (6p) Describe/explain concepts:
   (a) (3p) What is Copy-on-Write (CoW), which part of the OS employs CoW and why is it useful? ***Answer*** *See lecture slides for Chapter 9.*
   (b) (3p) Describe multilevel feedback queue scheduling. ***Answer*** *See lecture slides for Chapter 6.*

3. (12p) Compare/discuss:
   (a) (6p) What are *spinlocks* and how are they implemented? Why are they not recommended on uni-processors, but useful on multiprocessors? ***Answer*** *See lecture slides for Chapter 5. See book chapter for more details.*
   (b) (6p) In the context of file system implementation, compare *contiguous allocation*, *linked allocation* and *indexed allocation*. Give at least one advantage and one drawback for each. ***Answer*** *See lecture slides for Chapter 12.*

4. (10p) Assuming demand paging with three (3) frames, and the following page reference string
   `3 2 4 1 4 3 1 2 4 2 3 4 1 3 4 3 1 2 3 2`
   Show the page table contents for every access and count the page faults for
   (a) (4p) a FIFO page replacement strategy, and for
   (b) (4p) an optimal replacement strategy.
   (c) (2p) Compare the results and the feasibility of the strategies.

   Answer *See lecture slides for Chapter 9 for examples. FIFO gives 10 total, OPT gives 7. FIFO is easy to implement, OPT is impossible, because it needs knowledge about the future.*

5. (8p) Consider the `ptf.c` program (next page) using `fork()` and POSIX pthreads (on Linux, kernel $\geq$2.6). The program is compiled into `a.out` and run with `"./a.out hello world!"`. Assuming no errors occur,

(a) (2p) Which lines can result in system calls? How about the `pthread_*` calls?

(b) (3p) How many processes and how many threads are created? Motivate.

(c) (3p) What does the program output? Motivate.

Hint: Be extra-careful with `execv(...)`.

Listing 1: ptf.c

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <unistd.h>
4
5  void *run(void *ptr)
6  {
7      execv("/bin/echo",ptr);
8      if(fork() == 0)
9          fprintf(stderr,"done");
10     return ptr;
11 }
12
13 int main(int argc, char **argv)
14 {
15     pthread_t thread[2];
16     fork();
17     pthread_create(&thread[0], NULL, run, (void *) &argv[0]);
18     pthread_create(&thread[1], NULL, run, (void *) &argv[1]);
19     pthread_join(thread[0],NULL);
20     pthread_join(thread[1],NULL);
21     return 0;
22 }
```

Answer *For a) 7, 8, 9, 16, 21: execv, fork, printf, return-from-main are generating system calls. The Pthread_ * calls might or might not - the POSIX PThreads is only an API specification; the implementation is up to each. In particular for Linux 2.6, the standard library implements create via a clone() system call.*
For b), keep in mind that execv replaces the current process. First 16 creates one more process, 17-18 both create two additional threads (each process already contains a thread), but only the first gets to run; the exec will not allow anything else. So we have: 2 processes, with 3 threads each.
For c), the two processes end up running one execv each. Echo will output whatever strings are present in the arguments; arg[0] is treated as the name of the program. So the output will be two lines, a random combination of "hello world!" and "world!"
All these should be known from Chapters 2, 3, 4 and the shell laboratory assignment.

6. (8p) The *dining-philosophers* problem is a classic synchronization problem you should be familiar with already. Consider the following solution (pseudo-code):

Listing 2: Dining-philosophers

```
semaphore chopstick[5]; /* each initialized with 1 */

/* code for the thread modelling philosopher i (1..5) */
do {
        wait(chopstick[i]);
        wait(chopstick[(i+1)%5]);
```

```
        /* eat for a while */
        ...
        signal(chopstick[i]);
        signal(chopstick[(i+1)%5]);
        /* think for a while */
        ...
} while(true);
```

(a) (4p) What is the problem with this simple solution?

(b) (4p) Improve the code (without breaking concurrency) so that you fix the above problem.

Answer *See lecture slides for Chapter 5. Also the book suggests solutions to this.*